Leveraging Change Intents for Characterizing and Identifying Large-Review-Effort Changes

Song Wang*
wangsong@eecs.yorku.ca
York University

Chetan Bansal
chetanb@microsoft.com
Microsoft Research

Nachiappan Nagappan
nachin@microsoft.com
Microsoft Research

Adithya Abraham Philip
adithyaphilip@gmail.com
Microsoft Research India

ABSTRACT

Code changes to software occur due to various reasons such as bug fixing, new feature addition, and code refactoring. In most existing studies, the intent of the change is rarely leveraged to provide more specific, context aware analysis.

In this paper, we present the first study to leverage change intent to characterize and identify Large-Review-Effort (LRE) changes regarding review effort—changes with large review effort. Specifically, we first propose a feedback-driven and heuristics-based approach to obtain change intents. We then characterize the changes regarding review effort by using various features extracted from change metadata and the change intents. We further explore the feasibility of automatically classifying LRE changes. We conduct our study on a large-scale project from `Microsoft` and three large-scale open source projects, i.e., Qt, Android, and OpenStack. Our results show that, (i) code changes with some intents are more likely to be LRE changes, (ii) machine learning based prediction models can efficiently help identify LRE changes, and (iii) prediction models built for code changes with some intents achieve better performance than prediction models without considering the change intent, the improvement in AUC can be up to 19 percentage points and is 7.4 percentage points on average. The tool developed in this study has already been used in `Microsoft` to provide the review effort and intent information of changes for reviewers to accelerate the review process.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; **Programming teams**.

*Work done while interning at Microsoft.

PROMISE'19, September 18, 2019, Recife, Brazil

ACM ISBN 978-1-4503-7233-6/19/09…$15.00
https://doi.org/10.1145/3345629.3345635

KEYWORDS

Code review, change intent, review effort, machine learning

ACM Reference Format:
Song Wang, Chetan Bansal, Nachiappan Nagappan, and Adithya Abraham Philip. 2019. Leveraging Change Intents for Characterizing and Identifying Large-Review-Effort Changes. In *The Fifteenth International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE'19), September 18, 2019, Recife, Brazil.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3345629.3345635

1 INTRODUCTION

Code changes to software occur due to various reasons such as bug fixing, feature addition, and code refactoring. Often different changes have different motivations on behalf of the developers. For example, it is possible that a code merge, refactoring change, new feature addition all have different risk profiles due to the fact that they are different types of work actions performed by developers. In the past, in the large body of existing work about change analysis [10, 18, 27, 30], changes are all considered to be uniform. It is our goal in the paper to investigate if all changes are equal or if using the change intent for an example scenario of effort prediction, i.e., predicting changes that require large review effort, would result in building better, more accurate context-aware models.

This paper presents the first study to leverage change intent [6, 41] to characterize and understand *Large-Review-Effort (LRE) changes*, i.e., changes that have more than two iterations of code review. Other changes are treated as *regular changes*. In this study, we characterize the uniqueness of the *LRE changes* by using various features collected from the change metadata and the change intents, and explore the feasibility of automatically identifying the *LRE changes* by building machine learning based classifiers.

First, for understanding the change intent, following existing studies [15, 16, 22, 23], we use heuristics to annotate changes with different change intents by analyzing the commit messages, e.g., changes made for fixing bugs are labeled as 'Bug Fix'. As revealed in existing studies [8, 29], software changes could be made for multiple purposes, e.g., a change could be made for correcting bugs and refactoring existing code at the same time. In this study, we also label changes with multiple intents. For obtaining accurate change intents, we propose a feedback-driven approach to design

and refine the heuristics. Our manual evaluation shows the heuristics for categorizing changes achieve an accuracy higher than 80%. By using the change intents, we further analyse the statistical difference between *regular changes* and *LRE changes*, e.g., the distribution and the rate of *LRE changes* with different change intents. Second, in order to characterize the changes, we have collected various features from the metadata of changes and change intents, such as the process features [18, 49, 50], author information and experience, and the Word2Vec [5, 28] features generated from the commit messages to represent the changes. Correlation analysis shows that more than 70% of the features are correlated with the code review effort of changes. Third, based on the collected features, we further explore the feasibility of building machine learning based prediction models to classify the changes with and without considering the change intents.

This paper makes the following contributions:

- We propose a feedback-driven and heuristics-based approach to classify code changes into different change intents accurately for understanding changes.
- We show that change intents have a strong correlation with the review effort of the changes and changes with some intents are more likely to have more review iterations.
- We explore the feasibility of leveraging machine learning models to identify *LRE changes* on one project from `Microsoft` (referred to as `Microsoft` *project*) and three open source projects, i.e., Qt, Android, and OpenStack. Experiment results suggest that machine learning based models can be used to identify *LRE changes*. Random Forest, which is the best prediction model in our experiments, achieves AUC values larger than 0.71 on each of the four projects.
- We show that code review effort prediction models built on changes with particular change intents achieve better performance than the general prediction models that do not consider the change intents. **The tool developed in this study has already been used in `Microsoft` to provide the review effort and intent information of changes for reviewers to accelerate the review process.**

The rest of this paper is organized as follows. Section 2 presents the background and motivation. Section 3 describes the methodology of our approach. Section 4 shows the experimental setup. Section 5 presents the evaluation results. Section 6 discusses open questions and the threats to the validity of this work. Section 7 presents the related studies. Section 8 concludes this paper.

2 BACKGROUND AND MOTIVATION

Code changes could be problematic, e.g., they may introduce quality issues such as bugs, improper implementations, and maintenance issues. As a consequence, reviewing them could require much more code review effort. This study focuses on exploring the code review effort of changes. Specifically, based on the consensus of developers from `Microsoft`, we define a *Large-Review-Effort (LRE) change* as a code change that has more than two iterations of code review, e.g., if a code change cannot pass the first iteration of code review, developers have to conduct a second iteration of code review and even more iterations until they resolve all the review suggestions posted by reviewers. Other changes are treated as *regular changes*.

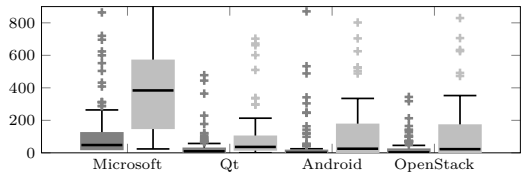


Figure 1: The distributions of review durations (hour) for *regular* and *LRE changes*. Gray bars (■) denote *regular* changes, light gray bars (■) denote *LRE* changes.

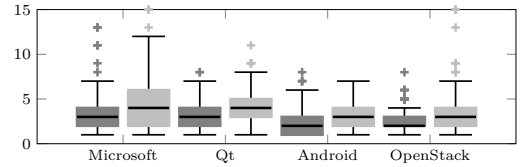


Figure 2: The distributions of #reviewers involved for reviewing changes. Gray bars (■) denote *regular* changes, light gray bars (■) denote *LRE* changes.

In this section, we motivate this study by showing the review effort of *regular* and *LRE* changes, i.e., review duration and the number of reviewers involved. Specifically, for each *regular change* and *LRE change* from the four projects in Table 1, we collect its duration in the code review system and the number of reviewers involved. A reviewer is involved if s/he is in the "Reviewers" field. We use the difference between the submission timestamp and the resolution timestamp of a review request of a change to assess its review duration. Note that we use the review duration to estimate the time effort of reviewing a change, since the exact time cost to review each change is not recorded in the code review systems. We then average the review duration and the number of reviewers involved for all changes for each project.

Figure 1 shows the distribution of review duration for each project. As shown in the figure, the average review duration of *LRE changes* could be 10X of that for *regular changes* (in project Android). Figure 2 shows the distribution of the number of reviewers involved to review both the *regular changes* and *LRE changes*. As we can see, on average reviewing the *LRE changes* requires more reviewers to collaborate together than reviewing the *regular changes* in each of the four projects.

We further conduct the Mann-Whitney U test ($p < 0.05$) to compare the differences of review duration and the number of involved between the two groups of changes. The results suggest that the review duration and the number of involved of *LRE changes* are significantly larger than that of *regular changes* respectively. Intuitively, finding *LRE changes* when they are submitted for code review, i.e., pre-merge and pre-deployment, can provide the review effort and intent information of changes for reviewers to accelerate the review process.

Table 1: Projects used in this study. #CR is the number of changes. LRERate is the rate of *LRE changes*.

Project	Lang	First Date	Last Date	#CR	LRERate (%)
Microsoft	C#	5/05/2015	5/22/2018	>100K	∼15
Qt	C	5/17/2011	5/25/2012	23,041	39.28
Android	JAVA	7/18/2011	5/31/2012	7,120	31.75
OpenStack	Python	7/18/2011	5/31/2012	6,430	43.31

3 APPROACH

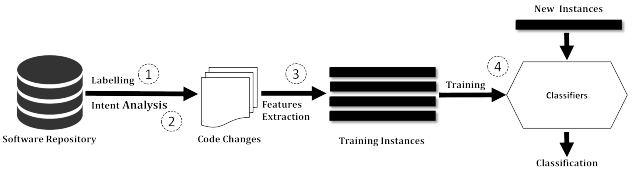


Figure 3: The overview of our *LRE change* prediction approach.

Figure 3 illustrates that our approach consists of four steps: (1) labeling each history change as a *regular change* or a *LRE change* (Section 3.1), (2) analyzing the change intents of all the changes (Section 3.2), (3) extracting features to represent the changes (Section 3.3), and (4) using the features and labels to build and train prediction models and then predicting new changes with the well-trained models (Section 3.4).

3.1 Labeling LRE Changes

The first step of our approach is to label each change as a *regular change* or a *LRE change* based on its code review history. Specifically, for the `Microsoft` project, we extracted its code review database, and checked the code review iteration count for each change, if it has more than two iterations of code review, we labeled it as a *LRE change* (based on the consensus of developers from `Microsoft`) otherwise we labeled it as a *regular change*. For the three open-source projects, since their code review systems do not maintain the code review iteration count, we use a heuristic approach to collect the *regular* and *LRE* changes. Specifically, in their code review system, a code review request of a change may have multiple iterations of code review, for each iteration, developers may submit a patchset to be reviewed, a patchset may have multiple patches. We counted the number of patchset for each code change. If the number of submitted patchsets is larger than two, we labeled the change as *LRE* otherwise we labeled it as *regular*.

3.2 Intent Analysis

Many approaches have been proposed to characterize and classify changes based on the change intents [1, 16, 33, 41]. Most of them consider the high-level change intents, e.g., corrective, adaptive, or perfective [41]. In this study, we leverage the fine-grained change intent categories proposed in Hindle et al. [16], which is shown in its Table 3, to categorize changes. Note that there are more than 20 different categories described in [16]. We started with manual analysis on randomly selected 200 changes from the four projects, and found that some of the categories have very few numbers of changes, e.g., 'Legal', 'Build', 'Branch' have less than three changes. We then group the small categories into larger ones for obtaining more instances. For example, we have grouped 'Legal', 'Data', 'Versioning', 'Platform Specific', and 'Documentation' into '**Resource**', grouped 'Rename' and 'Token Replace' into '**Refactor**', etc. Finally, we use eight types of change intents to categorize changes. Note that we also have an '**Other**' category for changes that do not fall into any of the eight categories.

Table 2 shows the nine types of changes, their descriptions, and the heuristics we used to automatically classify changes. Instead of manually labeling changes, in this work we automate the classification process by using well-refined heuristics. Thus, the accuracy of heuristics could significantly affect the result of this study. To improve the accuracy of the classification of changes, we used a feedback-driven approach to design and refine the heuristics for each type of change intents and the details are as follows:

Step 1: With a randomly selected 200 instances, the first two authors first classify them into the nine categories manually and independently. Specifically, after reading the commit message and checking the changed files of a change, they label the change based on their experience. For the classification conflicts (less than 5%), the third author inspects them independently and the first three authors make a decision for each conflict together. Then they initialize the heuristics for each category. We use $Cohen's Kappa$[1] to measure the inter-coder reliability of Step 1 and the score is 0.91.

Step 2: With the initialized heuristics, we classify all changes into at least one of the nine categories. For each category, we randomly collect 50 instances and the authors work together to check its accuracy manually.

Step 3: If the accuracy of a category is lower than 80%, we further refine the heuristics and then redo **Step 2**, otherwise we keep the heuristics for classifying changes.

Taking the '**Test**' category as an example, in **Step 1**, we found that most changes from it have keywords "test" or "testing" in their commit messages. Thus, the initialized heuristics we designed for '**Test**' is that the commit message of a change contains the keywords "test". In **Step 2**, we randomly checked 50 of the collected changes labeled as '**Test**'. Our manual inspection revealed that almost half of them were false positives, and we also found that most of the false positives have irrelevant commit messages, e.g., "... use backup config if test fails ..." and "... send a test message ...". To improve the heuristics, in **Step 3**, we added another heuristic, which is the changed files can only be test files (e.g., file names or paths contain the keyword "test") or resource files. Then we redo **Step 2** again, by using the new heuristics, the accuracy of '**Test**' category is around 90%.

We use the above steps to refine the heuristics of each category to ensure the classification of change intents has higher accuracy. Table 2 shows the final heuristics.

3.3 Feature Extraction

In this study, we use the following features for building machine learning based *LRE change* prediction models.

[1] https://en.wikipedia.org/wiki/Cohen%27s_kappa

Table 2: Heuristics for categorizing changes.

Change Intent	Description	Heuristics
Bug Fix	changes are made to fix bugs	1. the commit message contains keywords: "bug" or "fix" AND 2. the commit message does not contain keywords: "test case" or "unit test"
Resource	changes are made to update non-source code resources, configurations, or documents	1. the commit message contains keywords: "conf" or "license" or "legal" OR 2. if no keyword is matched in step 1, the changed files do not involve any source/test files
Feature	changes are made to implement new or update existing features	1. the commit message contains keywords: "update" or "add" or "new" or "create" or "add" or "implement feature" OR 2. changes in the 'Other' category that contain keywords: "enable" or "add" or "update" or "implement" or "improve"
Test	changes are made to add new or update existing test cases	1. the commit message contains the keyword: "test" OR 2. the changed files contain only test files or resource files
Refactor	changes are made to refactor existing code	the commit message contains the keyword: "refactor"
Merge	changes are made to merge branches	the commit message contains keywords: "merge" or "merging" or "integrate" or "integrated" or "integrated"
Deprecate	changes are made to remove deprecated code	the commit message contains keywords: "deprecat" or "delete" or "clean up"
Auto	changes that are committed by automated accounts or bots	the change is submitted by automated accounts or bots
Others	changes that are not in any of the above categories	-

Change Intent: As code changes could be classified into different categories based on their intents, we assume that changes with different intents have different impacts on the review effort of changes. We use a vector to represent the change intents of a change. Each element in the vector is a binary value, i.e., 1 or 0, representing whether the change has that intent or not. The change intents we considered are listed in Table 2.

Revision History: As presented in previous research [20], the revision history of a file can be a factor to predict its quality. In this study, we also explore the impact of revision history on predicting *LRE changes*. Specifically, given a change, we collect the number of files in this change that have been revised in the last 30 and 90 days, and the number of revision on all the involved files of this change in the last 30 and 90 days.

Owner Experience: This set of features represent the experience of a change's committer. We use a committer's commit history information to represent her/his experience, which includes the total number of changes submitted, the total number of *LRE changes* submitted, and the rate of submitted *LRE changes*. We assume that the committer's experience affects the review effort of the changes s/he submitted.

Word2Vec Features: Word embedding is a feature learning technique in natural language processing where individual words are no longer treated as unique features, but represented as a d-dimensional vector of real numbers that capture their contextual semantic meanings [28]. We train the embedding model by using all data from each project. With the trained word embedding model, each word can be transformed into a d-dimensional vector where d is set to 100 as suggested in previous studies [47]. Meanwhile a code change can be transformed into a matrix in which each row represents a term in its commit message. We then transform the code change matrix into a vector by averaging all the word vectors the code change contains, as described in [47].

Process Features: Various process features have been shown to help predict software bugs [18, 34]. In this study, we use the following process features: code addition, code deletion, number of changed files, and the types of changed files. Note that for the types of changed files, following existing studies [16, 17], we group files into source files, test files, configuration files, scripts, documentations, and others based on their suffixes and file paths. Specifically, we consider files with extensions: .java, .cs, .py, .js, .c, .cpp, .cc, .cp, .cxx, .c++, .h, .hpp, .hh, .hp, .hxx, and .h++, as the source files. Among them, files that contain 'test' in the paths or file names are considered as test files. Files with extensions: .script, .sh, .bash are considered as scripts. Files with extensions: .xml, .conf, .MF are considered as configuration files. Files with extensions: .htm, .html. .css, .txt, are considered as documentation files. The left files are considered as others.

Metadata: In addition to the above features, we also use metadata features of changes. Specifically, given a code change, we collect its commit minute (0, 1, 2, ... , 59), commit hour (0, 1, 2, ... , 23), commit day in a week (Sunday, Monday, ... , Saturday), commit day in a month (0, 1, 2, ... , 30), commit month in a year (0, 1, 2, ... , 11), and source file/path names.

All the features we used are available when the changes are submitted into the code review system.

3.4 Building Models and Predicting LRE Changes

After we obtain the features for changes, we split the data into the training and test datasets. We build and train the machine learning based prediction models on the training dataset and evaluate their performance on the test dataset. Following existing studies [15, 19, 34], we use 10-fold cross-validation to evaluate the prediction models.

4 EXPERIMENT SETUP

4.1 Research Questions

RQ1: What are the distributions of *LRE* and *regular* changes regarding change intents?

RQ2: Is it feasible to predict *LRE changes* by using machine learning based classifiers with features extracted from changes?

RQ3: Do the specific prediction models (classifiers trained on changes with a particular intent) outperform the general models (classifiers trained on all changes)?

RQ4: Does the performance of predicting *LRE changes* with a single intent differ from that of predicting *LRE changes* with multiple intents?

In RQ1, we aim at understanding the distributions of changes regarding the change intents. In RQ2, we explore the feasibility of predicting *LRE changes*. In RQ3, we aim to explore whether prediction models built on changes with particular intents can generate better performance. In RQ4, we investigate the difference in predicting *LRE changes* with a single intent and changes with multiple intents.

4.2 Experiment Data

To address our research questions, we perform empirical studies on software projects that actively adopt the code review process. We begin with the review dataset of Android, Qt, and OpenStack provided by Hamasaki et al. [12]. The three projects adopt the Gerrit[2] code review system. We also expand the review dataset to include code review data from a large-scale proprietary project from `Microsoft`. It adopts a custom code review system, which shares a similar review process with Gerrit. Details of the projects are in Table 1.

4.3 Evaluation Measures

To measure the performance of predicting *LRE changes*, we use four metrics: *Precision*, *Recall*, *F*1, and *AUC*. These metrics are widely adopted to evaluate prediction tasks [7, 18, 42, 46, 48, 50, 53]. Precision and recall are composed of three numbers regarding *true positive*, *false positive*, and *false negative*. True positive is the number of predicted *LRE changes* that are truly *LRE changes*, while false positive is the number of predicted *LRE changes* that are *regular changes*. False negative records the number of predicted *regular changes* that are *LRE changes*. F1 considers both precision and recall.

AUC is the area under the ROC curve, which measures the overall discrimination ability of a classifier. A machine learning model is considered applicable to classify a given dataset if the AUC score is larger than 0.7. It has been widely used to evaluate classification algorithms in prediction tasks [34, 43].

5 RESULTS AND ANALYSIS

5.1 RQ1: Distribution of Changes

Following the change intent taxonomy approach described in Section 4.1, we automatically label each change from the four projects. As reported in existing studies [8, 29], software changes could be made for multiple purposes, e.g., a change could be made for correcting bugs and refactoring existing code at the same time. Thus, we also label changes with multiple intents. Table 3 shows the number of changes, the percentage of changes with a particular change intent among all changes, and the percentage of *LRE changes* for each change intent in the four projects. In addition, we also show the numbers of changes that have single and multiple intents. Note that the '**Auto**' changes only exist in `Microsoft` project and we find all of them are regular changes, thus we exclude these changes for building change prediction models. Since there exist overlaps among different change intents, the sum of percentages of change intents is larger than 100.

As shown in Table 3, the distribution of changes regarding intents varies in different projects. We can see that changes are unevenly distributed regarding the intents. For example, changes with intents '**Bug Fix**' and '**Resource**' are dominating across the four projects, i.e., they take up more 50% of all the changes, while the percentages of changes with intents '**Refactor**' and '**Merge**' are less than 4%. Note that the distribution in our dataset is consistent with that of manually categorized changes from existing study [16], in which changes under categories **Corrective** (i.e., addressing failures), **Adaptive** (i.e., changes for data and processing environment), and **Perfective** (i.e., addressing inefficiency, performance, and maintainability issues) are dominating with a percentage higher than 60%, which also confirms the effectiveness of our automated heuristic-based change intent classification (details are presented in Section 3.2).

While '**Feature**' and '**Refactor**' have higher rates of *LRE changes*, this is reasonable since both the '**Feature**' and '**Refactor**' introduce new functionalities or restructure existing code snippets, which are easy to be problematic and require more code review effort. Category '**Resource**' has a lower rate across the four projects. This may be because, compared to all other categories, changes in the '**Resource**' category modify the source code rarely.

> Software code changes are unevenly distributed regarding change intents. Changes with some change intents, i.e., '**Feature**' and '**Refactor**', have a higher probability to be *LRE changes*.

5.2 RQ2: Feasibility of Predicting LRE Changes

This question explores whether machine learning algorithms can learn models that identify *LRE changes* among the submitted new changes under review. We use off-the-shelf machine learning algorithms from Weka [11] to build classification models. The used features include change intent, change history, owner experience, Word2Vec features, process features and metadata features (details are in Section 3.3).

Following existing studies [15, 18, 50, 53], we experiment with five widely used classifiers, i.e., Alternating Decision Tree (ADTree), Logistic Regression (Logistic), Naive Bayes (NB), Support Vector Machine (SVM), and Random Forest (RF). Note that this work does not intend to find the best-fitting classifiers or models, but to explore the feasibility of identifying *LRE changes* by using machine learning algorithms. Existing work [43] showed that selecting optimal parameter settings for machine learning algorithms could achieve better performance, thus we tune each of the classifiers with various parameters and use the ones that could achieve the best AUC value as our experiment settings. For each project, we build classification models and use the commonly used

[2] https://www.gerritcodereview.com/

Table 3: Taxonomy of code changes. #Change is the number of code changes. Percent is the percentage of changes with a particular change intent among all the changes. LRERate is the rate of *LRE changes*, which is measured in a percentage. Single contains changes that have only one intent. Multiple contains changes that have multiple intents. For confidentiality reasons, we did not release the numbers from `Microsoft`.

Change Intent	Microsoft		Qt			Android			OpenStack		
	Percent	LRERate	#Change	Percent	LRERate	#Change	Percent	LRERate	#Change	Percent	LRERate
Bug Fix	∼20	19.11	9,042	39.24	35.43	2,143	30.10	35.88	3,380	52.57	46.95
Resource	∼39	8.98	5,326	23.12	28.56	1,561	21.92	29.66	2,300	35.77	34.26
Feature	∼12	33.55	3,526	15.30	55.05	1,735	24.37	46.63	771	12.00	60.83
Test	∼4	15.14	3,840	16.67	38.75	663	9.31	35.6	1,005	15.63	54.23
Refactor	∼2	41.97	696	3.02	49.28	117	1.64	50.43	195	3.03	65.64
Merge	∼4	14.67	217	0.94	35.02	245	3.44	13.06	31	0.48	48.39
Deprecate	∼6	18.52	3,826	16.61	36.96	752	10.56	37.9	905	14.07	44.75
Auto	∼10	0	/	/	/	/	/	/	/	/	/
Others	∼15	20.96	4,036	17.52	40.21	1,513	21.25	35.23	577	8.97	34.66
Single	∼87	17.12	17,200	74.65	40.69	5,802	81.50	38.12	4,209	65.47	41.29
Multiple	∼13	14.71	5,841	25.35	35.13	1,317	18.50	32.88	2,220	34.53	47.07

Table 4: Comparison of different classifiers on predicting *LRE changes*. The best F1 and AUC values are in bold.

Project	ADTree		Logistic		NB		SVM		RF	
	F1	AUC	F1	AUC	F1	AUC	F1	AUC	F1	AUC
Microsoft	0.40	0.65	0.37	0.72	0.37	0.72	0.41	0.63	**0.46**	**0.76**
Qt	0.53	0.62	0.54	0.72	0.41	0.66	0.34	0.59	**0.57**	**0.71**
Android	0.58	0.68	0.54	0.71	0.58	0.70	0.50	0.65	**0.58**	**0.74**
OpenStack	0.60	0.64	0.64	0.76	0.62	0.68	0.66	0.70	**0.66**	**0.76**

10-fold cross-validation method to evaluate the prediction models [15, 18, 19, 34].

Table 4 shows the F1 and AUC values of each machine learning algorithm on the four experimental projects. Overall, of the five classifiers, RF consistently outperforms the others on each project. The improvements of RF compared to the other four classifiers range from 5.0 percentage points to 13.0 percentage points in AUC and 5.0 percentage points to 9.0 percentage points in F1. RF achieves similar AUC values on both the `Microsoft` project and open source projects, while it has a significantly lower F1 score (Wilcoxon signed-rank test, $p < 0.05$) on the `Microsoft` project. This may be because the `Microsoft` project has a lower rate of *LRE changes*, e.g., as shown in Table 1, the *LRE* rates of open-source projects are around 20.0 percentage points higher than that of the `Microsoft` project, which makes the data distribution more unbalanced in the `Microsoft` project. Previous studies showed that the unbalance issue could decline the F1 scores [43]. As revealed in existing work [43], the unbalance issue of a dataset does not impact the AUC measure and they suggested that a machine learning model is considered applicable to classify a given dataset if the AUC is larger than 0.7. Hence, we use the AUC to compare prediction models. We could find that among the five examined machine learning classifiers, two of them, i.e., Logistic and RF, achieve AUC values larger than 0.7 on each of the four experimental projects, which confirms the feasibility of identifying *LRE changes* by using machine learning algorithms.

> Machine learning based prediction models can help predict *LRE changes*. The best model (i.e., RF) achieves AUC values larger than 0.71 on each experimental project.

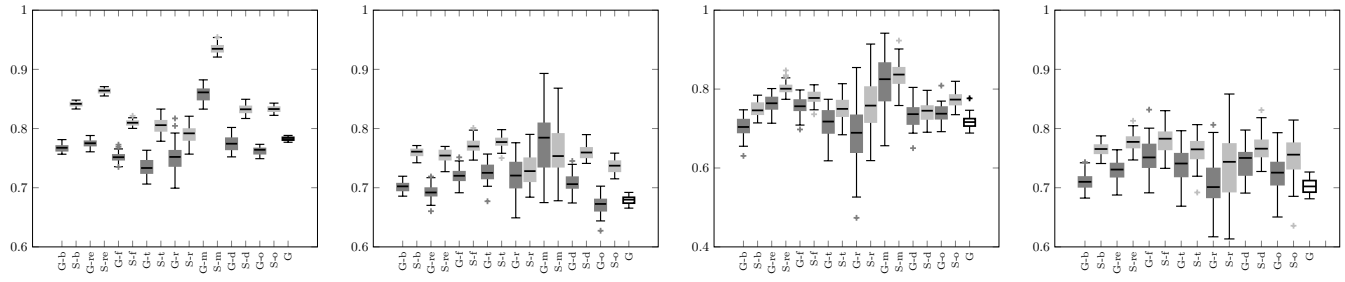
5.3 RQ3: Specific Models vs. General Models

In **RQ2**, we show that it is feasible to leverage machine learning classifiers to identify *LRE changes*. In this RQ, we further explore whether the machine learning classifiers built and trained on changes with a particular change intent, i.e., specific model, could achieve better performance than machine learning classifiers built and trained on all changes, i.e., general model. Specifically, for each project, we build and train the RF-based specific prediction models on changes with one particular change intent. We tune each of the RF-based classifiers with various parameter values and use the ones that could achieve the best AUC value as our experiment settings. In addition, we use 10-fold cross-validation method to evaluate the prediction models. The general model on the project is trained and evaluated on all changes without considering the change intents. Note that we exclude the specific model for category '**Merge**' on the project OpenStack, because it has very few numbers of instances.

Table 5 shows the comparison between the performance of the specific prediction models and the general models. Regarding F1, we can see that at least half of specific models outperform the general models across the four experimental projects. For example, six out of the eight specific models on the `Microsoft` project generate better F1 values than the general model, the improvement is up to 25.0 percentage points and is 6.0 percentage points on average. We observe a similar situation on Qt and OpenStack, i.e., overall specific models are better than the general models, the improvements are up to 16.0 and 15.0 percentage points on Qt and OpenStack respectively. However, we also observe an exception in Android, although five out of the eight specific models generate better (or the same) F1 values than the general model, the overall improvement is negative, the reason is that the '**Merge**' category has an F1 value that is 28.0 percentage points lower than that of the general model. This is because the '**Merge**' category has a much lower *LRE* rate (i.e., 13.1%) than that of all other categories (range from 29.0% to 50.4%) in Android, which makes the '**Merge**' unbalanced. Regarding AUC, we can observe that all the specific models outperform the general models across the four experimental

Table 5: Comparison between machine learning classifiers built on changes having a particular intent and machine learning classifiers built on all changes (i.e., general). Numbers in parenthesis are the differences between the specific models and the general models. Better F1 scores or AUC values that are larger than that of the general models are highlighted in bold. Note that we exclude the specific model for 'Merge' on the project OpenStack, since it only has 31 instances, which is not enough for training a machine learning classifier.

Change Intent	Microsoft				Qt				Android				OpenStack			
	P	R	F1	AUC	P	R	F1	AUC	P	R	F1	AUC	P	R	F1	AUC
Bug Fix	0.68	0.40	**0.51**(+0.05)	**0.84**(+0.08)	0.70	0.47	0.57(+0.00)	**0.77**(+0.06)	0.64	0.49	0.55(-0.03)	**0.76**(+0.02)	0.72	0.66	**0.69**(+0.03)	**0.79**(+0.03)
Resource	0.48	0.24	0.32(-0.14)	**0.79**(+0.03)	0.68	0.39	0.50(-0.07)	**0.76**(+0.05)	0.67	0.51	0.58(+0.00)	**0.82**(+0.08)	0.68	0.55	0.61(-0.05)	**0.81**(+0.05)
Feature	0.69	0.55	**0.61**(+0.15)	**0.81**(+0.05)	0.73	0.73	**0.73**(+0.16)	**0.78**(+0.07)	0.67	0.67	**0.67**(+0.09)	**0.77**(+0.03)	0.77	0.85	**0.81**(+0.15)	**0.81**(+0.05)
Test	0.61	0.22	0.32(-0.14)	**0.82**(+0.06)	0.71	0.55	**0.62**(+0.05)	**0.79**(+0.08)	0.62	0.53	0.58(+0.00)	0.74(+0.00)	0.74	0.77	**0.76**(+0.10)	**0.80**(+0.04)
Refactor	0.70	0.62	**0.65**(+0.19)	**0.79**(+0.03)	0.70	0.64	**0.67**(+0.10)	**0.76**(+0.05)	0.70	0.68	**0.69**(+0.11)	**0.76**(+0.02)	0.75	0.81	**0.78**(+0.12)	**0.78**(+0.02)
Merge	0.86	0.60	**0.71**(+0.25)	**0.95**(+0.19)	0.58	0.51	0.55(-0.02)	**0.77**(+0.06)	0.50	0.22	0.30(-0.28)	**0.81**(+0.07)	/	/	/	/
Deprecate	0.66	0.37	**0.47**(+0.01)	**0.84**(+0.08)	0.68	0.51	**0.58**(+0.01)	**0.77**(+0.06)	0.66	0.56	**0.61**(+0.03)	**0.78**(+0.04)	0.72	0.65	**0.69**(+0.03)	**0.80**(+0.04)
Others	0.68	0.50	**0.58**(+0.12)	**0.83**(+0.07)	0.65	0.51	0.57(+0.00)	**0.74**(+0.03)	0.60	0.51	0.55(-0.03)	**0.76**(+0.02)	0.62	0.51	0.56(-0.10)	**0.78**(+0.02)
General	0.51	0.41	0.46	0.76	0.60	0.54	0.57	0.71	0.61	0.55	0.58	0.74	0.68	0.65	0.66	0.76



(a) Microsoft **(b) Qt** **(c) Android** **(d) OpenStack**

Figure 4: Comparison of perdition performance (AUC) between the specific models and the general models. The model with a prefix "G" means using the trained general model to predict changes with a particular intent. "b" represents 'Bug Fix', "re" represents 'Resource', "f" represents 'Feature', "t" represents 'Test', "r" represents 'Refactor', "m" represents 'Merge', "d" represents 'Deprecate', and "o" represents 'Others'. For example G-b means using the trained general model to predict changes with the 'Bug Fix' intent. S-b is the specific model trained and evaluated on changes with the 'Bug Fix' intent.

projects. The improvement could be up to 19.0 percentage points and is 7.4 percentage points on average. Thus, from the comparison shown in Table 5, we conclude that overall the specific models achieve better prediction performance than the general models.

Above all, we show that the specific models (built on changes with a particular change intent) are overall better than the general models (built on all the changes). One could also argue that using the general models to predict changes with a particular intent may have better performance than the corresponding specific model. To explore this issue, we further examine the performance of leveraging the general models to predict changes with a particular intent. Specifically, for each change intent, we randomly divide its changes into training dataset and test dataset (2/3 for training, 1/3 for test) following existing studies [14, 24]. For the specific model, we train the model on the training data and evaluate its performance on the test dataset. For the corresponding general model, we combine the training data from all specific models, and evaluate its performance on the test dataset of a specific model. We repeat the data splitting, model training, and evaluation 50 times to reduce bias.

Figure 4 shows the boxplots of the 50 times classification for each specific model and its corresponding general model on each project. In addition, we also show the boxplots of overall-general models (i.e., using 2/3 of all changes to train the models and evaluate the models on the left 1/3 changes without considering change intents). Each boxplot presents the AUC distribution (median and upper/lower quartiles) of a prediction model. We use gray (●), light gray (●), and white (○) to represent the specific models, corresponding general models, and the overall-general models respectively. We could observe that overall the specific models outperform the general models on almost all the change intents across the four experimental projects. Specifically, for the `Microsoft` project, the mean AUC values of the specific models are around ten percentage points higher than that of the corresponding general models. For the open source projects, the mean AUC values of the specific modes are around five percentage points higher than that of the corresponding general models. In addition, all the specific models outperform the overall-general models.

> Code review effort prediction models built on changes with particular change intents achieve better performance than the general prediction models that do not consider the change intents. Thus, we suggest to build specific models for better *LRE changes* prediction.

5.4 RQ4: Single Intent vs. Multiple Intents

As shown in RQ1 (Section 5.1), in this study we labeled changes with multiple intents. This RQ explores the performance of *LRE changes* prediction models on changes with a single intent and changes with multiple intents. Specifically,

Table 6: Performance of predicting *LRE changes* with single and multiple change intents. Better F1 or AUC values are in bold.

Change Intent	Microsoft		Qt		Android		OpenStack	
	F1	AUC	F1	AUC	F1	AUC	F1	AUC
Single	**0.48**	**0.78**	**0.58**	**0.75**	**0.58**	**0.73**	**0.67**	**0.80**
Multiple	0.41	0.75	0.54	0.70	0.57	0.71	0.66	0.77

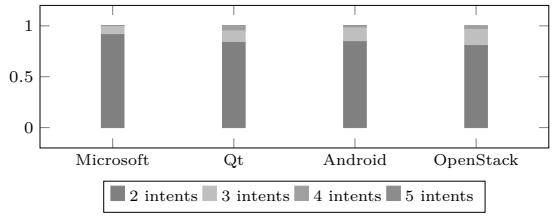


Figure 5: The distribution of changes with multiple change intents.

for each project, we build and train the RF-based prediction models with changes with only a single intent and changes with multiple intents respectively. We tune each of the RF-based classifiers with various parameter values and use the ones that could achieve the best AUC value as our experiment settings. We also use the 10-fold cross-validation method to evaluate the models.

Table 6 shows the F1 scores and AUC values of the prediction models for changes with single and multiple change intents in the four experimental projects. As we can see, the prediction models for changes with a single intent significantly outperform the prediction models for changes with multiple intents in both F1 and AUC across the four experimental projects (Wilcoxon signed-rank test, $p < 0.05$). In terms of F1, the improvement could be up to 7.0 percentage points and is 3.3 percentage points on average. For AUC, the improvement could be up to 5.0 percentage points and is 3.3 percentage points on average. One of the possible reasons for this difference is that changes with a single intent are mainly made for one specific purpose, which makes them easier to be distinguished by machine learning classifiers than complex changes with multiple intents.

Machine learning based classifiers generate better performance on changes with a single change intent than changes with multiple change intents.

6 DISCUSSION

6.1 LRE Rate Analysis

As shown in Table 3, in three of the four projects, changes with a single intent have a higher *LRE* rate than that of changes with multiple intents. To explore the reason behind this phenomenon, we first break down the number of changes with multiple intents from the four projects, which are shown in Figure 5. We could observe that among the changes with multiple intents, changes that have two intents are dominating, i.e., 90% of the changes with multiple intents involve only two different intents. Thus, we narrow down the analysis to explore the distribution of changes with two change intents. To do this, we collect the changes with two intents from the four projects and count the number of different intent combinations among these changes. Figure 6 shows the top ten types of changes with two intents, which cover more than 97% of all the changes with two intents.

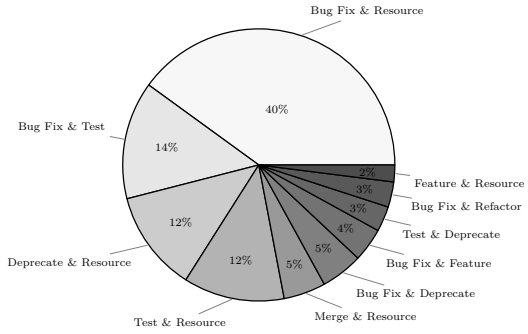


Figure 6: The distribution of code changes with multiple intents.

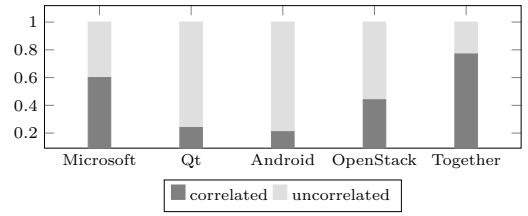


Figure 7: The distributions of correlated and uncorrelated features in each project and across the projects (i.e., Together).

As we could see that 'Bug Fix' & 'Resource', 'Bug Fix' & 'Test', 'Deprecate' & 'Resource', and 'Test' & 'Resource' are the dominating combinations. With the distribution chart in Figure 6, we infer the reasons why changes with multiple intents have lower *LRE* rates as follows. First, each intent may represent an independent task and developers may modify source code for each intent separately, which provides them more opportunities to check the modified code and eventually improve the quality of the change. Second, some of the combined intents represent the standard software quality assurance process, which can help improve the quality of the changes. Taking changes with double intents 'Bug Fix' & 'Test' as an example, developers may first fix a bug and then modify the test cases to validate the fix of the bug, which makes the changes more reliable.

6.2 Feature Correlation Analysis

Following existing studies [9], we use the Spearman rank correlation [51] to compute the correlations between the metrics described in Section 3.3 and the review effort of changes, i.e., *regular changes* or *LRE changes*. Values greater than 0.10 can be considered a small effect size; values greater than 0.30 can be considered a medium effect size [52]. In this work, we consider the values larger than 0.10 or smaller than -0.10 as correlated, others are uncorrelated.

Figure 7 shows the distributions of correlated and uncorrelated features in each project and across the four projects. As we could observe, in the `Microsoft` project, the percentage

of correlated features (around 60%) is larger than that of the open source projects (less than 50%). One of the possible reasons is that compared to the open source projects, the `Microsoft` project has a larger experimental dataset, e.g., around 5X of the size of open source projects, which provides sufficient data to evaluate each feature and reduce the potential bias. Overall more than 70% features are correlated with review effort of changes, i.e., *regular changes* or *LRE changes*. We further examined the selected features and found that they covered five different feature types, i.e., change intents, revision history, owner experience, Word2Vec features, and process features, while none of the metadata features is selected as correlated. This observation suggests that the commit time of a change does not affect its review effort.

The Spearman correlation analysis shows that most of the collected features are correlated with the review effort of changes, thus the collected features are applicable for identifying *LRE changes*.

6.3 Threats to Validity

Internal Validity. The main threat to internal validity is about the annotation of change intents, subjectivity of annotation, and miscategorization. The annotation relied on our manually refined heuristics, and although this approach is a common practice, this process contains bias since the authors of this paper are not the developers of these projects. To mitigate this, authors worked independently to annotated the data and refined the heuristics. In addition, we chose a setup that ensures that every heuristic is cross-validated and the classification conflicts have to pass the third inspection. In this work, we define *Large-Review-Effort (LRE) changes* based on the consensus of developers from `Microsoft`, the performance of our approach may vary with different thresholds.

External Validity. In this work, we use a project from `Microsoft` and three open source projects to evaluate our proposed approach. Since they adopt different code review systems, i.e., the `Microsoft` project adopts a custom code review system and the open source projects adopt the Gerrit code review system. Thus, the proposed approach might not work for projects that adopt other code review systems.

7 RELATED WORK

7.1 Change Intent Analysis

In order to understand the change intents of code changes, Swanson [41] first proposed a classification of maintenance activities as corrective, adaptive, and perfective. Along this line, many change analysis models have been proposed [6, 13, 16, 22, 23, 39, 40]. Buckley et al. [6] proposed a taxonomy of software evolution to characterize the mechanisms of changes. Lehnert et al. [22, 23] proposed comprehensive investigations of software change impact analysis. Later, Hassan et al. [13] extended Swanson's categorization by adding three new categories, i.e., bug fixing changes, general maintenance changes, and feature introduction changes. Hindle et al. [16] extended Swanson's categorization with two new categories, i.e., feature addition and non-functional. Hassan's categories are not specific enough, which only provided high-level information of categories. Hindle's extended categories contain more detailed types of changes for each category. In this study we adopted the fine-grained change type information provided Hindle's work [16].

7.2 Software Code Review

Code review is a manual inspection of source code by humans, which aims at identifying potential defects and quality problems in the source code before its deployment in a live environment [2–4, 25, 26, 31, 32, 44, 45]. Many studies have examined the practices of code review. Stein et al. [38] explored the distributed, asynchronous code inspections. Laitenburger [21] surveyed code inspection methods, and presented a taxonomy of code inspection techniques. In recent years, Modern Code Review (MCR) has been developed as a tool-based code review system and becomes popular and widely used in both commercial software (e.g., Google [37], Microsoft) and open-source software (e.g., Android, Qt, and OpenStack) [12]. Rigby et al. [35, 36] have done extensive work examining code review practices in OSS development. Bacchelli & Bird find that understanding of the code and the reason for a change is the most important factor in the quality of code reviews [2].

In this paper, we explore the feasibility of accelerating code review by identifying the code changes that require multiple rounds of code review.

8 CONCLUSION

This paper presents the first study of *LRE changes* in code review system, i.e., changes that have more than two iterations of code review by using the change intents. We conduct our study on one large-scale project from `Microsoft`, and three open source projects, i.e., Qt, Android, and OpenStack. Experiment results show that: (i) changes with some intents are more likely to be reviewed with multiple iterations, (ii) machine learning based prediction models could help identify *LRE changes*, and (iii) prediction models built for changes with some intents achieve better performance than prediction models without considering the intents. The tool developed in this study has already been used in `Microsoft` to provide the review effort and intent information of changes for reviewers to accelerate the review process.

In the future, we plan to conduct real-world case studies to explore how much our approach can accelerate code review process. In addition, our work on change intents is just the first step in a large body of work. We would like to explore if change intent improves the fidelity and accuracy of other prediction tasks, e.g., code reviewer recommendation, software defect prediction, and effort estimation.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback which helped improve this paper.

REFERENCES

[1] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. 2008. What's a typical commit? a characterization of open source software repositories. In *ICPC'08*. 182–191.

[2] Alberto Bacchelli and Christian Bird. 2013. Expectations, outcomes, and challenges of modern code review. In *ICSE'13*. 712–721.

[3] Olga Baysal, Oleksii Kononenko, Reid Holmes, and Michael W Godfrey. 2013. The influence of non-technical factors on code review. In *WCRE'13*. 122–131.

[4] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. 2014. Modern code reviews in open-source projects: Which problems do they fix?. In *MSR'14*. 202–211.

[5] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A Neural Probabilistic Language Model. *The Journal of Machine Learning Research* 3 (2003), 1137–1155.

[6] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. 2005. Towards a taxonomy of software change. *Journal of Software Maintenance and Evolution: Research and Practice* 17, 5 (2005), 309–332.

[7] Qiang Cui, Song Wang, Junjie Wang, Yuanzhe Hu, Qing Wang, and Mingshu Li. 2017. Multi-Objective Crowd Worker Selection in Crowdsourced Testing. In *SEKE'17*. 1–6.

[8] Ying Fu, Meng Yan, Xiaohong Zhang, Ling Xu, Dan Yang, and Jeffrey D Kymer. 2015. Automated classification of software change messages by semi-supervised Latent Dirichlet Allocation. *IST'15* 57 (2015), 369–377.

[9] Emanuel Giger, Martin Pinzger, and Harald C Gall. 2012. Can we predict types of code changes? an empirical analysis. In *MSR'12*. 217–226.

[10] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. 2000. Predicting fault incidence using software change history. *TSE'00* 26, 7 (2000), 653–661.

[11] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. 2009. The WEKA data mining software: an update. *ACM SIGKDD explorations newsletter* 11, 1 (2009), 10–18.

[12] Kazuki Hamasaki, Raula Gaikovina Kula, Norihiro Yoshida, AE Cruz, Kenji Fujiwara, and Hajimu Iida. 2013. Who does what during a code review? datasets of oss peer review repositories. In *MSR'13*. 49–52.

[13] Ahmed E Hassan. 2008. Automated classification of change messages in open source projects. In *SAC'08*. 837–841.

[14] Kim Herzig, Sascha Just, Andreas Rau, and Andreas Zeller. 2013. Predicting defects using change genealogies. In *ISSRE'13*. 118–127.

[15] Abram Hindle, Daniel M German, Michael W Godfrey, and Richard C Holt. 2009. Automatic classication of large changes into maintenance categories. In *ICPC'09*. 30–39.

[16] Abram Hindle, Daniel M German, and Ric Holt. 2008. What do large commits tell us?: a taxonomical study of large commits. In *MSR'08*. 99–108.

[17] Abram Hindle, Michael W Godfrey, and Richard C Holt. 2007. Release pattern discovery via partitioning: Methodology and case study. In *MSR'07*. 19.

[18] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *ASE'13*. 279–289.

[19] Sunghun Kim, E James Whitehead Jr, and Yi Zhang. 2008. Classifying software changes: Clean or buggy? *TSE'08* 34, 2 (2008), 181–196.

[20] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting faults from cached history. In *ICSE'07*. 489–498.

[21] Oliver Laitenberger. 2002. A survey of software inspection technologies. In *Handbook of Software Engineering and Knowledge Engineering: Volume II: Emerging Technologies*. 517–555.

[22] Steffen Lehnert. 2011. A taxonomy for software change impact analysis. In *IWPSE-EVOL'11*. 41–50.

[23] Steffen Lehnert, Matthias Riebisch, et al. 2012. A taxonomy of change types and its application in software evolution. In *ECBS'12*. 98–107.

[24] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *TSE'08* 34, 4 (2008), 485–496.

[25] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2014. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *MSR'14*. 192–201.

[26] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering* 21, 5 (2016), 2146–2189.

[27] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. 2011. Local vs. global models for effort estimation and defect prediction. In *ASE'11*. 343–351.

[28] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *NIPS'13*. 3111–3119.

[29] Audris Mockus and Lawrence G Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *ICSM'00*. 120.

[30] Audris Mockus and David M Weiss. 2000. Predicting risk of software changes. *Bell Labs Technical Journal* 5, 2 (2000), 169–180.

[31] Rodrigo Morales, Shane McIntosh, and Foutse Khomh. 2015. Do code review practices impact design quality? a case study of the qt, vtk, and itk projects. In *SANER'15*. 171–180.

[32] Adam Porter, Harvey Siy, and Lawrence Votta. 1996. A Review of Software Inspections. *Advances in Computers* 42 (1996), 39–76.

[33] Ranjith Purushothaman and Dewayne E Perry. 2005. Toward understanding the rhetoric of small source code changes. *TSE'05* 31, 6 (2005), 511–526.

[34] Foyzur Rahman and Premkumar Devanbu. 2013. How, and why, process metrics are better. In *ICSE'13*. 432–441.

[35] Peter C Rigby, Daniel M German, and Margaret-Anne Storey. 2008. Open source software peer review practices: a case study of the apache server. In *ICSE'08*. 541–550.

[36] Peter C Rigby and Margaret-Anne Storey. 2011. Understanding broadcast based peer review on open source software projects. In *ICSE'11*. 541–550.

[37] Caitlin Sadowski, Emma Söderberg, Luke Church, Michal Sipko, and Alberto Bacchelli. 2018. Modern code review: a case study at google. In *ICSE-SEIP'18*. 181–190.

[38] Michael Stein, John Riedl, Sören J Harner, and Vahid Mashayekhi. 1997. A case study of distributed, asynchronous software inspection. In *ICSE'97*. 107–117.

[39] Xiaobing Sun, Bixin Li, Chuanqi Tao, Wanzhi Wen, and Sai Zhang. 2010. Change impact analysis based on a taxonomy of change types. In *COMPSAC'10*. 373–382.

[40] Xiaobing Sun, Bixin Li, Wanzhi Wen, and Sai Zhang. 2013. Analyzing impact rules of different change types to support change impact analysis. *SEKE'13* 23, 03 (2013), 259–288.

[41] E Burton Swanson. 1976. The dimensions of maintenance. In *ICSE'76*. 492–497.

[42] Xinye Tang, Song Wang, and Ke Mao. 2015. Will this bug-fixing change break regression testing?. In *ESEM'15*. 1–10.

[43] Chakkrit Tantithamthavorn, Ahmed E Hassan, and Kenichi Matsumoto. 2019. The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *TSE'19* (2019).

[44] Patanamon Thongtanunam, Shane McIntosh, Ahmed E Hassan, and Hajimu Iida. 2016. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *ICSE'16*. 1039–1050.

[45] Lawrence G Votta Jr. 1993. Does every inspection need a meeting? *FSE'93* 18, 5 (1993), 107–114.

[46] Junjie Wang, Qiang Cui, Song Wang, and Qing Wang. 2017. Domain adaptation for test report classification in crowdsourced testing. In *ICSE-SEIP'17*. 83–92.

[47] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images Don't Lie: Duplicate Crowdtesting Reports Detection With Screenshot Information. *IST'19* (2019).

[48] Junjie Wang, Song Wang, and Qing Wang. 2018. Is there a golden feature set for static warning identification?: an experimental evaluation. In *ESEM'18*. 17.

[49] Song Wang, Taiyue Liu, Jaechang Nam, and Lin Tan. 2018. Deep semantic feature learning for software defect prediction. *TSE'18* (2018).

[50] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *ICSE'16*. 297–308.

[51] Arnold D Well and Jerome L Myers. 2003. *Research design & statistical analysis*. Psychology Press.

[52] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *ICST'10*. 421–428.

[53] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *PROMISE'07*. 9–9.